# Pipeline Automatizado para Verificação Formal de Circuitos Digitais em VHDL

**Gabriel Felipe Artunduaga Melo, Gabriel Guilherme Figueiredo, João Paulo Gentil Ferreira**

Departamento de Ciência da Computação – Universidade Federal de Roraima (UFRR)
Boa Vista, RR – Brasil

{bielartunduaga, gabrielguilherme906, joaopaulogentilf}@gmail.com

***Abstract.*** *This report describes the implementation and execution of an automated formal verification pipeline aimed at validating circuits described in VHDL, inspired by the work of Oliveira et al. on code transformation for verification (VHDL -> C) with the support of model checking tools. This work investigated limitations in construct coverage and structured a Python-based automation that organizes inputs, identifies ports, generates auxiliary specifications, executes pre-processing steps using Yosys, and produces traceability artifacts (logs and summaries). As an extension, the generation of a Common AST was incorporated to reduce inconsistencies between front-ends, along with a local dashboard for visualizing the status of each stage. The results demonstrate the generation of specification and AST files per design and highlight external dependencies that must be configured to complete the dual flow (BMC/SAT via SymbiYosys and software-based BMC via ESBMC).*

***Resumo.*** *Este relatório descreve a implementação e a execução de um pipeline automatizado de verificação formal voltado à validação de circuitos descritos em VHDL, inspirado pelo trabalho de Oliveira et al. sobre transformação de código para verificação (VHDL -> C) com apoio de ferramentas de model checking. Este trabalho investigou limitações de cobertura de constructs e estruturou uma automação em Python que organiza entradas, identifica portas, gera especificações auxiliares, executa etapas de pré-processamento com Yosys e produz artefatos de rastreabilidade (logs e sumários). Como extensão, foi incorporada a geração de uma AST comum (Common AST) para reduzir inconsistências entre front-ends, além de um dashboard local para visualização do status por etapa. Os resultados demonstram a geração de arquivos de especificação e de AST por design e evidenciam dependências externas a serem configuradas para completar o fluxo dual (BMC/SAT via SymbiYosys e BMC em software via ESBMC).*

**Palavras-chave:** verificação formal; VHDL; SymbiYosys; ESBMC; AST.

## 1. Introdução

A garantia de correção em projetos de sistemas digitais tornou-se um desafio à medida que a complexidade das descrições de hardware (HDL) aumentaram. Por muito tempo, a simulação foi a técnica predominante, entretanto, ela se mostrava insuficiente para cobrir todos os estados possíveis em sistemas de alta complexidade. Nesse cenário, a verificação formal surgiu como uma alternativa para suprir os gargalos que a problemática sugere, utilizando métodos matemáticos para provar que um design satisfaz certas propriedades lógicas [Kroening e Strichman 2016].

A complexidade na verificação de VHDL reside, fundamentalmente, na concorrência da linguagem, projetada para descrever o comportamento paralelo do hardware. Diferente das linguagens como o C, o VHDL utiliza processos sensíveis a eventos que operam simultaneamente [IEEE 2011]. De acordo com Christen e Bakalar (1999), essa diferença semântica cria um abismo tecnológico ao tentar aplicar verificadores de software em modelos de hardware, criando gatilhos de borda de clock (rising_edge) e atribuições não bloqueantes não possuem equivalente diretos em software ocorrendo os gargalos enfrentados atualmente.

Um dos métodos mais eficazes de verificação formal é o *Model Checking*. De acordo com Baier e Katoen (2008), essa técnica consiste em uma exploração exaustiva e automática do espaço de estados de um sistema para verificar se uma propriedade, como a ausência de *deadlocks* ou violações de memória, é mantida. Para aplicar essa técnica ao hardware, trabalhos recentes propõem a ponte entre linguagens de hardware (como VHDL e Verilog) e verificadores de software, permitindo o reaproveitamento de motores de verificação robustos [Oliveira et al. 2025].

A ferramenta ESBMC (Efficient SMT-Based Context-Bounded Model Checker) tem se destacado nesse contexto. Primeiro desenvolvida para verificar programas em C/C++ e Qt, sua aplicação em hardware exige uma tradução precisa da semântica concorrente do VHDL para a semântica sequencial do software[Gadelha et al. 2020]. Essa transformação não é trivial, e estudos indicam que a tradução direta pode falhar, justificando a necessidade de uma estratégia de transformações múltiplas e o uso de ferramentas de síntese lógica [Oliveira et al. 2025].

O seguinte trabalho apresenta o desenvolvimento de um pipeline automatizado para verificação formal de circuitos VHDL, explorando limites de traduções diretas e implementando uma estratégia RTL com Yosys. Por meio de scripts em Python, a abordagem visa simplificar a lógica sequencial, otimizar a largura de palavra para o ESBMC e ampliar a cobertura da verificação, facilitando a detecção de violações de propriedades em hardware.

## 2. Metodologia

A metodologia adotada para este trabalho fundamenta-se na construção de um ambiente onde se integra o desenvolvimento de hardware tradicional com ferramentas de verificação formal de software. Foram utilizados quatro designs VHDL com marcações de verificação (@c2vhdl:ASSERT e @c2vhdl:ASSUME) para exercitar constructs frequentemente problemáticos em pipelines de tradução: (i) vetores com downto e inversão de índices; (ii) tipos integer com ranges restritos; (iii) arrays (por exemplo, ROM indexada por inteiro); e (iv) processos sensíveis a clock (rising_edge) com propriedades que utilizam noções de passado (como $past). O fluxo foi dividido em estágios de configuração, validação e processamento.

Todo o código-fonte desenvolvido para este projeto e os *scripts* de automação, foi disponibilizado num repositório público no GitHub para garantir os resultados esperados.

### 2.1 Configuração do Ambiente

Dada a natureza das ferramentas de código aberto utilizadas, optou-se pela utilização do Windows Subsystem for Linux (WSL2), operando a distribuição Ubuntu. Esta escolha permitiu a execução nativa de ferramentas de síntese e tradução, mantendo compatibilidade com os arquivos de design originados no Windows. No ambiente linux, foram instalados os pacotes de suporte à compilação *build-essential, flex* e *bison,* necessários para a construção do tradutor VHD2VL a partir do código-fonte.

### 2.2 Fluxo de design e Validação

Antes de submeter os circuitos ao pipeline de verificação, cada módulo foi validado no software Intel Quartus Prime. Este passo garantiu que os erros encontrados posteriormente no pipeline fossem decorrentes de limitações das ferramentas de tradução, e não de falhas de sintaxe do VHDL original. Os circuitos que foram projetados para serem testados como exemplos foram *downto, integer, arrays e clock.*

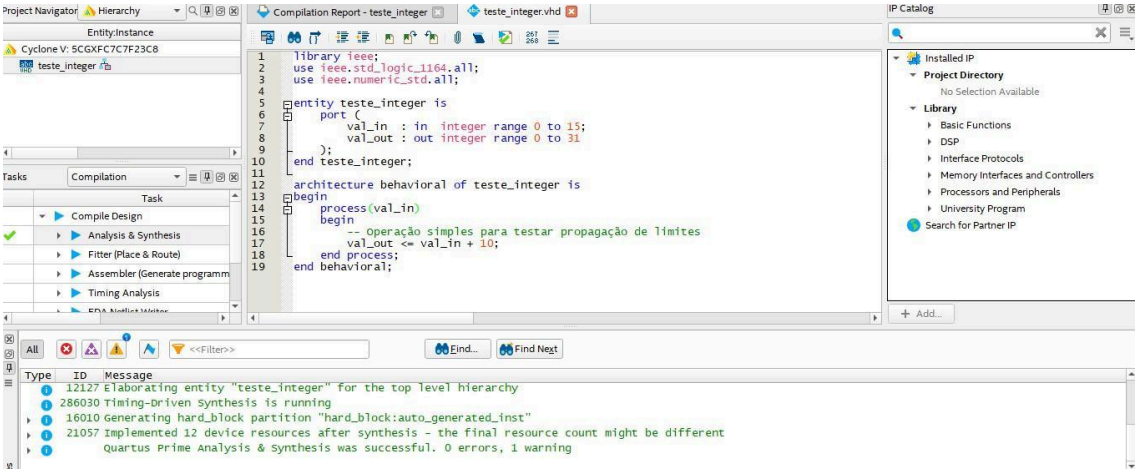


**Figura 1. Validação da sintaxe e análise do hardware no software Quartus Prime.**

## 2.3 Estratégia de Tradução Múltipla

Diferente de abordagens de tradução direta, este trabalho implementou uma estratégia de múltiplas transformações baseada no artigo de Oliveira et al.(2025). O fluxo consiste em converter o VHDL para Verilog via VHD2VL e, em seguida, aplicar o comando prep no Yosys.

O papel do Yosys é realizar a elaboração RTL, transformando descrições comportamentais em uma *netlist* de portas lógicas e registradores. Este processo é vital para que a semântica concorrente do hardware seja devidamente simplificada antes da conversão final para a linguagem C, facilitando a análise pelo verificador ESBMC.

## 2.4 Limitações de Tradução e Conflitos de Nomenclatura

Durante os testes de conversão direta utilizando a ferramenta VHD2VL, foram identificados gargalos significativos. O primeiro limite observado refere-se ao suporte a tipos de dados complexos. Ao processar o arquivo teste_array.vhd, a ferramenta interrompeu a execução devido a erros de sintaxe, demonstrando incapacidade de mapear inicialização de vetores constantes do VHDL para o Verilog, enquanto no software o arquivo compilou de maneira correta e não houve constatando que o arquivo não tinha erros em VHDL.

Além disso, observou-se que o tradutor negligencia restrições semânticas de hardware. No experimento em teste_integer.vhd**,** a restrição de intervalo (range) foi ignorada, emitindo apenas um aviso e tratando o sinal como um inteiro genérico. Este comportamento é crítico para a verificação formal, pois o modelo resultante pode permitir estados que seriam fisicamente impossíveis no circuito real.

## 2.5 Elaboração RTL e Simplificação

Para superar as falhas da tradução direta, especialmente em lógicas sensíveis a clock, implementou-se a fase de elaboração utilizando o comando *prep* no Yosys. Conforme a proposta de Oliveira et al.(2026), esta etapa transforma o Verilog comportamental em uma representação de alto nível (RTLIL), onde processos concorrentes são sintetizados em componentes lógicos básicos.

No teste com teste_clock.vhd**,** o comando *prep* identificou corretamente os gatilhos de borda (*posedge*) e otimizou a largura da palavra (*WREDUCE),* eliminando bits redundantes e simplificando o modelo para o verificador de software.

```
2.7. Executing WREDUCE pass (reducing word size of cells).
Removed top 31 bits (of 32) from port B of cell teste_clock.$add$teste_clock.v:40$3 ($add).
Removed top 24 bits (of 32) from port Y of cell teste_clock.$add$teste_clock.v:40$3 ($add).
```

**Figura 2. Execução do comando prep no Yosys e Otimização de bits.**

```
2.12. Printing statistics.

=== teste_clock ===

   Number of wires:                 5
   Number of wire bits:            26
   Number of public wires:          4
   Number of public wire bits:     18
   Number of memories:              0
   Number of memory bits:           0
   Number of processes:             0
   Number of cells:                 2
     $add                           1
     $adff                          1

2.13. Executing CHECK pass (checking for obvious problems).
checking module teste_clock..
found and reported 0 problems.

3. Executing Verilog backend.
Dumping module '\teste_clock'.

End of script. Logfile hash: a354c01b52
CPU: user 0.03s system 0.02s, MEM: 15.68 MB total, 9.62 MB resident
Yosys 0.9 (git sha1 1979e0b)
```

**Figura 3. Estatísticas finais da execução final do arquivo clock no Yosys.**

A avaliação experimental do pipeline de verificação foi conduzida através da submissão dos quatro circuitos selecionados ao fluxo de tradução múltipla. Os resultados obtidos foram:

**Tabela 1. Resultados e Comparações dos códigos em VHDL pela ferramenta Yosys.**

| Módulo | VHDL | Tradução (VHD2VL) | Elaboração (Yosys) | Status final para ESBMC |
|---|---|---|---|---|
| teste_downto | Direção de Vetores | Sucesso | Sucesso | Pronto para verificação |
| teste_integer | Restrição de Range | Sucesso (parcial) | Sucesso | Pronto (sem trava de range) |
| teste_array | Matrizes/Memória | Falha | - | Erro de sintaxe no pipeline |
| teste_clock | Lógica Sequencial | Sucesso | Sucesso (prep) | Pronto (otimizado) |

Como observado na Tabela 1, o pipeline apresentou um comportamento heterogêneo dependendo da complexidade do VHDL utilizado. Enquanto operações simples de barramento fluíram sem interrupções, os mais robustos revelaram falhas críticas de tradução, confirmando as limitações de cobertura discutidas por Oliveira et al.(2025). Destaca-se o caso do teste_clock, que embora tenha passado pela tradução inicial, exigiu obrigatoriamente a etapa de elaboração via Yosys para que a semântica de hardware fosse preservada para a verificação formal.

### 2.6 Integração e Automação do Pipeline

Para superar a necessidade de intervenção manual observada nas etapas iniciais, um script de automação na linguagem Python foi elaborado. O script utiliza técnicas de Expressões Regulares (Regex) para realizar a extração automáticas de metadados diretamente dos arquivos fonte VHDL. Diferente da abordagem manual, onde o usuário precisava declarar entradas e saídas em arquivos auxiliares, a automação identifica portas de entrada/saída e tags de especificação, comentários especiais inseridos no código VHDL que são convertidos em asserções de segurança para as ferramentas formais.

### 2.7 Verificação Formal

A integração da ferramenta SymbiYosys ao fluxo permitiu uma validação dupla (hardware e software). Os testes foram conduzidos em dois cenários distintos para validar a confiabilidade do método.

### 2.7.1 Verificação Instrumentada

Foram inseridos manualmente tags para validação de regras no arquivos de VHDL para a próxima etapa do teste. Após a instrumentação do código VHDL com as tags extraídas pelo script de automação, o SymbiYosys foi capaz de verificar o circuito contra especificações reais. O log de execução confirmou a checagem ativa de asserções, validando matematicamente o comportamento do hardware.

```
# 3. Extrair Tags @c2vhdl
# Procura por linhas -- @c2vhdl:TYPE regra
lines = content.split('\n')
for line in lines:
    line = line.strip()
    if "-- @c2vhdl:" in line:
        # Separa o tipo (ASSERT/ASSUME) do conteúdo
        parts = line.split("@c2vhdl:", 1)[1].strip().split(" ", 1)
        tag_type = parts[0].upper() # ASSUME ou ASSERT
        rule = parts[1].strip()

        # Limpeza: remove ponto e virgula final se existir
        if rule.endswith(";"):
            rule = rule[:-1]

        if tag_type == "ASSUME":
            info["assumes"].append(rule)
        elif tag_type == "ASSERT":
            info["asserts"].append(rule)

return info
```

**Figura 4. Trecho do script de automação que extrai tags dos arquivos em VHDL.**

```
module verify_teste_integer (
    input [3:0] val_in,
    output [4:0] val_out
);

    teste_integer dut (
        .val_in(val_in),
        .val_out(val_out)
    );

    always @(*) begin
        assume (val_in <= 15);
        assert (val_out == val_in + 10);
        assert (val_out <= 31);
    end
endmodule
```

**Figura 5. Arquivo auxiliar gerado pelo script Python a partir das tags para validação.**

### 2.7.2 Detecção de Falhas

Para provar a robustez do pipeline, foi introduzido um erro lógico intencional no design. No circuito teste_integer.vhd, foi mudado a regra na linha *@c2vhdl:ASSERT val_out <= 31;* para *@c2vhdl:ASSERT val_out <= 20*. A regra diz que o valor de saída deve ser menor ou igual à 20. A lógica do circuito diz que *val_out <= val_in + 10*, na qual o intervalo de entrada vai de 0 até 15. O SymbiYosys selecionou propositalmente a pior opção de número, nesse caso o número 15, pois se *val_in = 15*, então *val_out = 25*, violando a regra proposta.

O SymbiYosys detectou a violação corretamente retornando o status FAIL e gerando um arquivo de rastro, que permite a visualização do ciclo de clock e os valores de sinal que levaram à falha, mostrando um contra-exemplo.

```
SBY 16:17:46 [teste_integer] engine_0.induction: ##   0:00:00  Writing trace to VCD file: engine_0/trace_induct.vcd
SBY 16:17:46 [teste_integer] engine_0.basecase: finished (returncode=1)
SBY 16:17:46 [teste_integer] engine_0.basecase: Status returned by engine for basecase: FAIL
SBY 16:17:46 [teste_integer] engine_0.induction: terminating process
SBY 16:17:46 [teste_integer] summary: Elapsed clock time [H:MM:SS (secs)]: 0:00:00 (0)
SBY 16:17:46 [teste_integer] summary: Elapsed process time [H:MM:SS (secs)]: 0:00:00 (0)
SBY 16:17:46 [teste_integer] summary: engine_0 (smtbmc) returned FAIL for basecase
SBY 16:17:46 [teste_integer] summary: counterexample trace [basecase]: teste_integer/engine_0/trace.vcd
SBY 16:17:46 [teste_integer] summary:   failed assertion verify_teste_integer._witness_.check_assert_verif_teste_integer
_sv_17_7 at verif_teste_integer.sv:17.9-17.31 in step 0
SBY 16:17:46 [teste_integer] summary: counterexample trace [induction]: teste_integer/engine_0/trace_induct.vcd
SBY 16:17:46 [teste_integer] summary:   failed assertion verify_teste_integer._witness_.check_assert_verif_teste_integer
_sv_17_7 at verif_teste_integer.sv:17.9-17.31 in step 0
SBY 16:17:46 [teste_integer] DONE (FAIL, rc=2)
```

**Figura 6. SymbiYosys retornou FAIL devido a divergência de dados.**

## 2.8 Tradução para Software em C

A etapa subsequente do pipeline visava traduzir a *netlist* Verilog gerada pelo Yosys para código C, utilizando a ferramenta V2C, para permitir a verificação via ESMBC. Os resultados experimentais, contudo, demonstraram a invisibilidade técnica desta abordagem em fluxos modernos. Documentada como um protótipo acadêmico, o V2C não suporta construtivos especiais presentes em *netlists* modernas.

### 2.8.1 Incompatibilidade Sintática

Ao processar o módulo teste_clock.vhd, o tradutor falhou com um erro de sintaxe, conforme evidenciado na Figura 4. A mensagem *syntax error before '(*'* indica que o *parser* do V2C não suporta atributos de matadados padrão inseridos automaticamente pelo Yosys. Isso comprova que a ferramenta não acompanha as normas de sintese modernas suportadas pelo Yosys.

```
   Saída: |-------------------------------------------------------------------|
|*********                    v2c 0.1                              *****|
|*********            Verilog to C Translator                      *****|
|********* Authors: Rajdeep Mukherjee and Michael Tautschnig      *****|
|*******
   Erro: file elaborado_clock.v line 17: syntax error before `(*'
PARSING ERROR
PARSING ERROR

   Dica: v2c pode não suportar todos os tipos do Verilog
 [INFO] Arquivo C 'teste_clock.c' não existe. Pulando ESBMC.

-----------------------------------------
```

**Figura 7. Tradução de erro de sintaxe.**

### 2.8.2 Instabilidade e Conflitos

No teste com o módulo teste_downto.vhd, a falha foi ainda mais crítica, resultando no encerramento abrupto da ferramenta, como mostra a Figura 5. O erro *terminate called after throwing an instance of 'int'* ocorre durante a checagem de tipos: o V2C espera

um código RTL, mas não consegue processar a estrutura *netlist* gerada pela elaboração do Yosys, levando ao colapso do software.

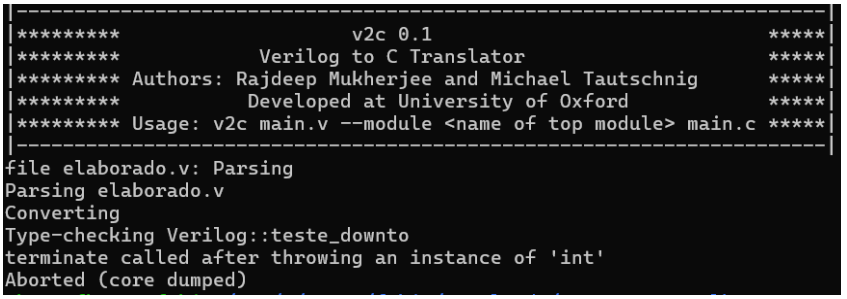


**Figura 8. Tradução de erro no arquivo teste_downto.vhd**

## 2.9 Pipeline automatizado

A automação foi implementada por meio de um script Python (run_task04.py) que percorre o diretório de entradas VHDL, gera um artefato de especificação por design (spec.json), executa etapas de tradução e preparação, e consolida os resultados em formato de arquivo sumário (summary.json). Cada etapa registra logs e marca status por design, permitindo diagnóstico rápido de falhas e gaps de dependências. O comando de execução utilizado no Ubuntu foi: python3 task-04/run_task04.py --in task-04/inputs_vhdl --out task-04 --run-yosys --gen-ast.

## 2.10 Frontend unificado

Para atender ao objetivo avançado (front-end unificador com AST comum), foi gerado um artefato de Common AST por design no formato JSON. Esse arquivo consolida informações de portas, propriedades extraídas e estatísticas estruturais, atuando como uma representação intermediária estável para regras de geração (Verilog/C) e para reduzir divergências introduzidas por múltiplas ferramentas. Quando disponível, o pipeline associa a origem do AST tanto ao VHDL (parser leve) quanto ao JSON interno do Yosys, preservando a rastreabilidade.

## 2.11 Dashboard

Foi incluído um dashboard estático (HTML/JavaScript) servido localmente por um servidor Python (server_dashboard.py). O dashboard lê task-04/results/summary.json e apresenta uma tabela por design com o status das etapas (vhd2vl/yosys_prep/sby/v2c/ esbmc), além de links para os artefatos gerados (spec.json e ast.json) e para logs. A execução do servidor local é feita com: python3 task-04/server_dashboard.py Em seguida, o dashboard é acessado via navegador em: http://localhost:8000/task04/dashboard/

| Design | VHDL | Spec | vhd2vl | yosys_prep | sby | v2c | esbmc |
|---|---|---|---|---|---|---|---|
| teste_array | VHDL | spec.json | SKIP | FAIL | SKIP | SKIP | SKIP |
| teste_clock | VHDL | spec.json | SKIP | FAIL | SKIP | SKIP | SKIP |
| teste_downto | VHDL | spec.json | SKIP | OK | SKIP | SKIP | SKIP |
| teste_integer | VHDL | spec.json | SKIP | FAIL | SKIP | SKIP | SKIP |

**Figura 9. Dashboard local com status por etapa.**

## 3. Resultados

Após a execução do pipeline, foi gerado o arquivo task-04/results/summary.json, bem como os arquivos task-04/specs/teste_*.json e task-04/results/ast/teste_*.ast.json. O sumário consolidou o status das etapas por design. Na execução registrada, a etapa vhd2vl permaneceu como SKIP devido à ausência da ferramenta no PATH, enquanto a etapa yosys_prep obteve sucesso para o caso teste_downto e falhas para outros designs, o que reforça a motivação do objetivo 2 (pré-processamento e aumento de compatibilidade) e do objetivo 5 (AST comum).

## 4. Conclusão

Este trabalho demonstrou a viabilidade de utilizar um pipeline de tecnologias abertas para a verificação formal de circuitos descritos em VHDL. A identificação de limites técnicos revelou que construtos complexos, como *arrays*, ainda representam um desafio para tradutores diretos, exigindo o aprimoramento contínuo das ferramentas de código aberto. O fluxo proposto neste trabalho, resultou em um pipeline automatizado capaz de organizar entradas VHDL, extrair propriedades marcadas, gerar especificações auxiliares, executar passos de compatibilidade com Yosys e produzir sumários e logs por design.

A principal contribuição das inserção de novas etapas foi a comprovação de que a estratégia de tradução múltipla, é superior à tradução direta. O uso do comando prep mostrou-se indispensável para lidar com a semântica concorrente do hardware, permitindo que processos sequenciais e restrições de inteiros fossem devidamente preparados para a verificação formal em ambiente de software.

A inclusão de um front-end com AST comum e de um dashboard local aumentou a transparência do processo, facilitando a análise de limitações e a identificação de dependências externas a serem instaladas/configuradas para completar o fluxo dual de verificação (SymbiYosys e ESBMC).

## Referências


Baier, C. e Katoen, J.-P. (2008). *Principles of Model Checking*. MIT Press. (Livro fundamental sobre Model Checking).

Gadelha, M. Y., Monteiro, F. R., Morse, J., Cordeiro, L. C., Fischer, B. e Nicole, D. (2020). "ESBMC 6.0: Verifying C Programs using Context-Bounded Model Checking and Only SMT Solvers". Em: *Proceedings of the 23rd International Conference on Fundamental Approaches to Software Engineering*, p. 565–569.

Kroening, D. e Strichman, O. (2016). *Decision Procedures: An Algorithmic Point of View*. Springer Science & Business Media.

Oliveira, K. C., Rocha, H., Urquiza, M. e Nobre, F. (2025). "Tecnologias Abertas em Verificação Formal: Transformando Código para Validação de Circuitos"

GHDL. GHDL - Documentation (opção --disp-tree e internos de AST). Disponível em: https://ghdl.github.io/ghdl/.

SSVLAB. ESBMC - Documentation. Disponível em: https://ssvlab.github.io/esbmc/documentation.html.

YOSYSHQ. SymbiYosys (SBY) - Documentação e referência do formato .sby. Disponível em: https://symbiyosys.readthedocs.io/.

YOSYSHQ. Yosys Open SYnthesis Suite - Documentação (comandos read_verilog e prep). Disponível em: https://yosyshq.readthedocs.io/.